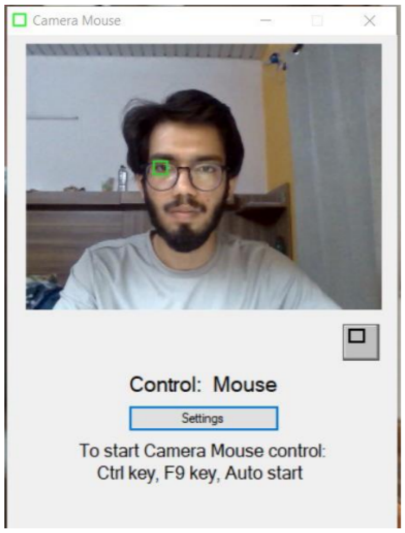
# Eye Tracking for Mouse Control in OpenCV



Let's adopt a baby-steps approach. The very first thing we need is to read the webcam image itself. You can do it through the `VideoCapture` class in the OpenCV `highgui` module. `VideoCapture` takes one parameter, the webcam index or a path to a video.

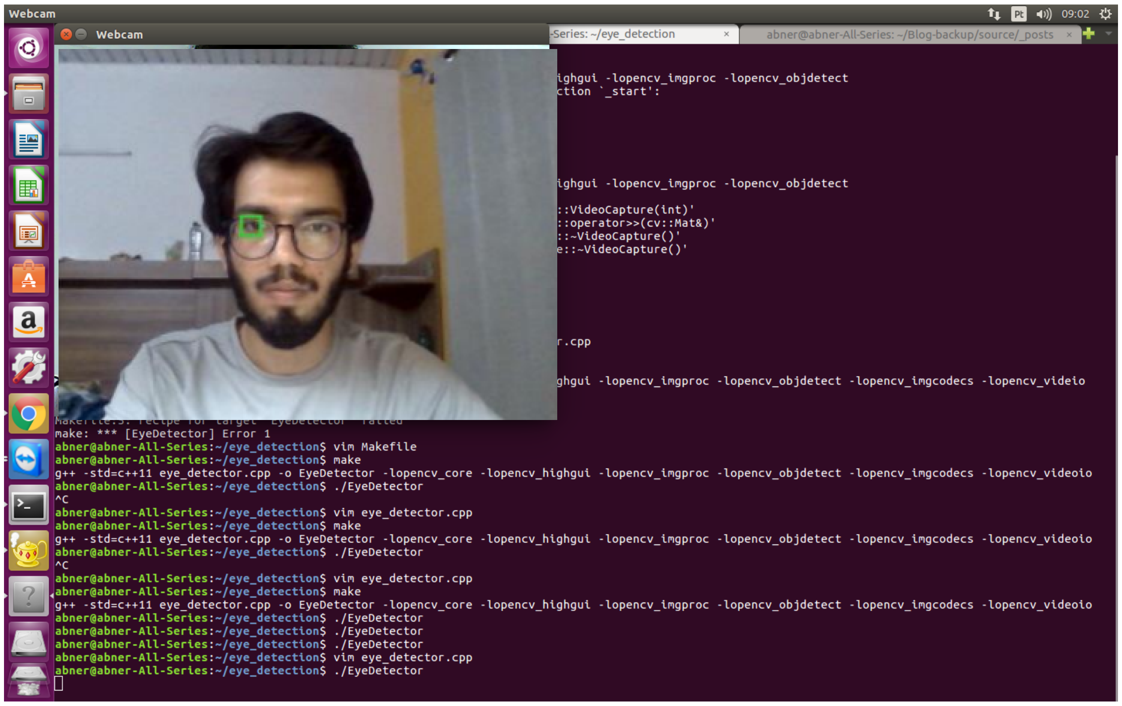
`eye_detector.cpp`

```cpp
#include <iostream>

#include <opencv2/core/core.hpp>
#include <opencv2/highgui/highgui.hpp>
#include <opencv2/imgproc/imgproc.hpp>
#include <opencv2/objdetect/objdetect.hpp>

int main()
{
  cv::VideoCapture cap(0); // the fist webcam connected to your PC
  if (!cap.isOpened())
  {
      std::cerr << "Webcam not detected." << std::endl;
      return -1;
  }
  cv::Mat frame;
  while (1)
  {
      cap >> frame; // outputs the webcam image to a Mat
      cv::imshow("Webcam", frame); // displays the Mat
      if (cv::waitKey(30) >= 0) break; // takes 30 frames per second. if the user presses
  }
  return 0;
}
```

I took the liberty of including some OpenCV modules besides the necessary because we are going to need them in the future.

Compile it with this Makefile:

```
Makefile
CPP_FLAGS=-std=c++11
OPENCV_LIBS: -lopencv_core -lopencv_highgui -lopencv_imgproc -lopencv_objdetect -lopencv_im
LD_FLAGS=$(OPENCV_LIBS)

default: EyeDetector
EyeDetector: eye_detector.cpp
  g++ $(CPP_FLAGS) $^ -o $@ $(LD_FLAGS)
clean:
  rm -f EyeDetector
```



```
ighgui -lopencv_imgproc -lopencv_objdetect
ction `_start':

ighgui -lopencv_imgproc -lopencv_objdetect

::VideoCapture(int)'
::operator>>(cv::Mat&)'
::~VideoCapture()'
e::~VideoCapture()'

r.cpp

ghgui -lopencv_imgproc -lopencv_objdetect -lopencv_imgcodecs -lopencv_videio

make: *** [EyeDetector] Error 1
abner@abner-All-Series:~/eye_detection$ vim Makefile
abner@abner-All-Series:~/eye_detection$ make
g++ -std=c++11 eye_detector.cpp -o EyeDetector -lopencv_core -lopencv_highgui -lopencv_imgproc -lopencv_objdetect -lopencv_imgcodecs -lopencv_videoio
abner@abner-All-Series:~/eye_detection$ ./EyeDetector
^C
abner@abner-All-Series:~/eye_detection$ vim eye_detector.cpp
abner@abner-All-Series:~/eye_detection$ make
g++ -std=c++11 eye_detector.cpp -o EyeDetector -lopencv_core -lopencv_highgui -lopencv_imgproc -lopencv_objdetect -lopencv_imgcodecs -lopencv_videoio
abner@abner-All-Series:~/eye_detection$ ./EyeDetector
^C
abner@abner-All-Series:~/eye_detection$ vim eye_detector.cpp
abner@abner-All-Series:~/eye_detection$ make
g++ -std=c++11 eye_detector.cpp -o EyeDetector -lopencv_core -lopencv_highgui -lopencv_imgproc -lopencv_objdetect -lopencv_imgcodecs -lopencv_videoio
abner@abner-All-Series:~/eye_detection$ ./EyeDetector
abner@abner-All-Series:~/eye_detection$ ./EyeDetector
abner@abner-All-Series:~/eye_detection$ ./EyeDetector
abner@abner-All-Series:~/eye_detection$ vim eye_detector.cpp
abner@abner-All-Series:~/eye_detection$ ./EyeDetector
```

Now you can see that it’s displaying the webcam image. That’s something!

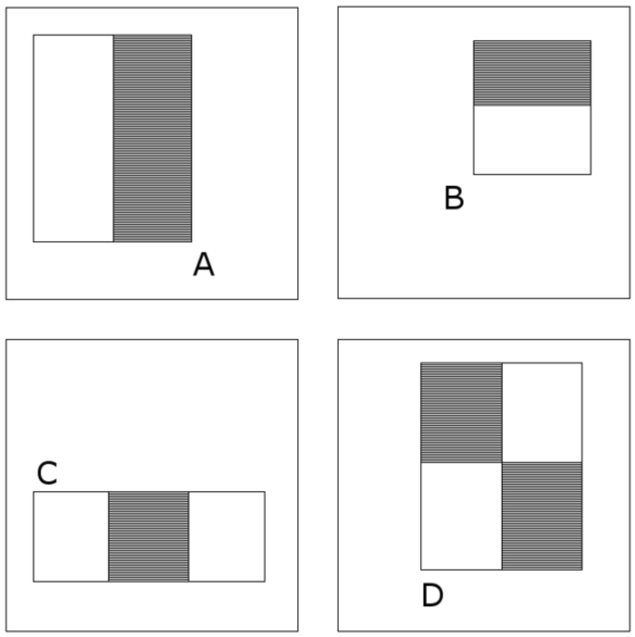
Now let’s get into the computer vision stuff!

## FACE AND EYE DETECTION WITH VIOLA-JONES ALGORITHM (THEORY)

Here’s a bit of theory (you can skip it and go to the next section if you are just not interested): Humans can detect a face very easily, but computers do not. When an image is prompted to the computer, all that it “sees” is a matrix of numbers. So, given that matrix, how can it predict if it represents or not a face? Answer: Building probability distribuitions through thousands of samples of faces and non-faces. And it’s the role of a **classifier** to build those probability distribuitions. But here’s the thing: A regular image is composed by thousands of pixels. Even a small 28x28 image is composed by 784 pixels. Each pixel can assume 255 values (if the image is using 8-bits grayscale representation). So that’s $255^{784}$ number of possible values. Wow! Estimate probability distribuitions with some many variables is not feasible. This is where the Viola-Jones algorithm kicks in: It extracts a much simpler representations of the image, and combine those simple representations into more high-level representations in a hierarchical way, making the problem in the highest level of representation much more simpler and easier than it would be using the original image. Let’s see all the steps of this algorithm.

### Haar-like Feature Extraction

We have some primitive “masks”, as shown below:



Those masks are slided over the image, and the sum of the values of the pixels within the “white” sides is subtracted from the “black” sides. Now the result is a **feature** that represents that region (a whole region summarized in a number).

### Weak classifiers

Next step is to train many simple classifiers. Each classifier for each kind of mask. Those simple classifiers work as follows: Takes all the features (extracted from its corresponding mask) within the face region and all the features outside the face region, and label them as “face” or “non-face” (two classes). It then learns to distinguish features belonging to a face region from features belonging to a non-face region through a simple **threshold function** (i.e., faces features generally have value above or below a certain value, otherwise it’s a non-face). This classifier itself is very bad and is almost as good as random guesting. But if combined, they can arise a much better and stronger classifier (weak classifiers, unite!)

### Cascading classifiers

Given a region, I can submit it to many weak classifiers, as shown above. Each weak classifier will output a number, 1 if it predicted the region as belonging to a face region or 0 otherwise. This result can be weighted. The sum of all weak classifiers weighted outputed results in another **feature**, that, again, can be inputted to another classifier. It’s said that that new classifier is a **linear combination** of other classifiers. Its role is to determine the right weight values such as the error be as minimum as possible.

### What about eyes?

Well, eyes follow the same principle as face detection. But now, if we have a face detector previously trained, the problem becomes sightly simpler, since the eyes will be always located in the face region, reducing dramatically our search space.

## FACE AND EYE DETECTION WITH VIOLA-JONES ALGORITHM (PRACTICE)

Thankfully, the above algorithm is already implemented in OpenCV and a classifier using thousands and thousands of faces was already trained for us!

Let’s start by reading the trained models. You can download them here. Put them in the same directory as the .cpp file.

eye_detector.cpp

```
int main()
{
  cv::CascadeClassifier faceCascade;
  cv::CascadeClassifier eyeCascade;
  if (!faceCascade.load("./haarcascade_frontalface_alt.xml"))
  {
      std::cerr << "Could not load face detector." << std::endl;
      return -1;
  }
  if (!eyeCascade.load("./haarcascade_eye_tree_eyeglasses.xml"))
  {
      std::cerr << "Could not load eye detector." << std::endl;
      return -1;
  }
  ...
}
```

Now let's modify our loop to include a call to a function named `detectEyes`:

`eye_detector.cpp`

```
int main()
{
  ...
  while (1)
  {
      ...
      detectEyes(frame, faceCascade, eyeCascade);
      cv::imshow("Webcam", frame);
      if (cv::waitKey(30) >= 0) break;
  }
  return 0;
}
```

Let's implement that function:

`eye_detector.cpp`

```
void detectEyes(cv::Mat &frame, cv::CascadeClassifier &faceCascade, cv::CascadeClassifier &
{
  cv::Mat grayscale;
  cv::cvtColor(frame, grayscale, CV_BGR2GRAY); // convert image to grayscale
  cv::equalizeHist(grayscale, grayscale); // enhance image contrast
  std::vector<cv::Rect> faces;
  faceCascade.detectMultiScale(grayscale, faces, 1.1, 2, 0 | CV_HAAR_SCALE_IMAGE, cv::Size(
}
```

A break to explain the `detectMultiScale` method. It takes the following arguments:

- inputImage: The input image
- faces: A vector of rects where the faces were detected
- scaleFactor: The classifier will try to upscale and downscale the image in a certain factor (in the above case, in 1.1). It will help to detect faces with more accuracy.
- minNumNeighbors: How many true-positive neighbor rectangles do you want to assure before predicting a region as a face? The higher this face, the lower the chance of detecting a non-face as face, but also lower the chance of detecting a face as face.
- flags: Some flags. In the above case, we want to scale the image.
- minSize: The minimum size which a face can have in our image. A poor quality webcam has frames with 640x480 resolution. So 150x150 is more than enough to cover a face in it.

Let's proceed. Now we have the faces detected in the vector `faces`. What to do next? Eye detection!

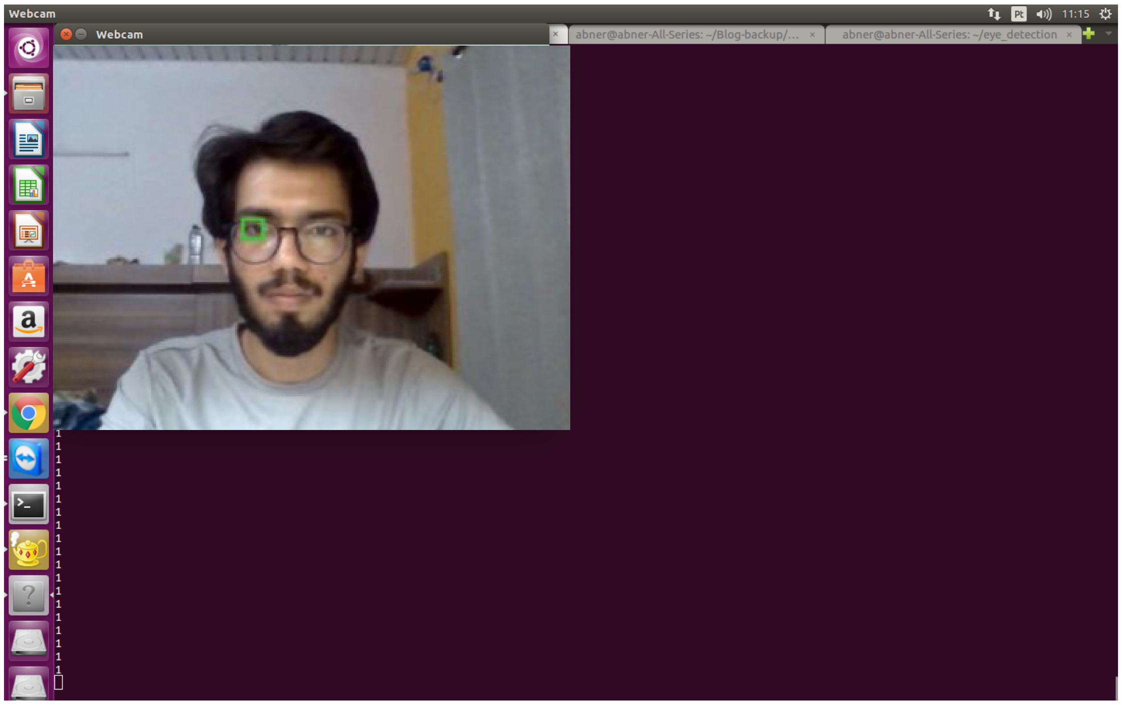
`eye_detector.cpp`

```
void detectEyes(...)
{
  ...
  if (faces.size() == 0) return; // none face was detected
  cv::Mat face = frame(faces[0]); // crop the face
  std::vector<cv::Rect> eyes;
  eyeCascade.detectMultiScale(face, eyes, 1.1, 2, 0 | CV_HAAR_SCALE_IMAGE, cv::Size(150, 15
}
```

Now we have both face and eyes detected. Let's just test it by drawing the regions where they were detected:

`eye_detector.cpp`

```
void detectEyes(...)
{
  ...
  rectangle(frame, faces[0].tl(), faces[0].br(), cv::Scalar(255, 0, 0), 2);
  if (eyes.size() != 2) return; // both eyes were not detected
  for (cv::Rect &eye : eyes)
  {
      rectangle(frame, faces[0].tl() + eye.tl(), faces[0].tl() + eye.br(), cv::Scalar(0, 2
  }
}
```


Webcam
11:15
Webcam
abner@abner-All-Series: ~/Blog-backup/...
abner@abner-All-Series: ~/eye_detection

```
1
1
1
1
1
1
1
1
1
1
1
1
1
1
1
1
1
1
1
```

Looking good so far!

## DETECTING IRIS

Now we have detected the eyes, the next step is to detect the iris. For that, we are going to look for the most "circular" object in the eye region. Luckily, that's already a function in OpenCV that does just that! It's called `HoughCircles`, and it works as follows: It first apply an edge detector in the image, from which it make contours and from the contours made it tried to calculate a "circularity ratio", i.e., how much that contour looks like a circle.

First we are going to choose one of the eyes to detect the iris. I'm going to choose the leftmost.

`eye_detector.cpp`

```cpp
cv::Rect getLeftmostEye(std::vector<cv::Rect> &eyes)
{
  int leftmost = 99999999;
  int leftmostIndex = -1;
  for (int i = 0; i < eyes.size(); i++)
  {
      if (eyes[i].tl().x < leftmost)
      {
          leftmost = eyes[i].tl().x;
          leftmostIndex = i;
      }
  }
  return eyes[leftmostIndex];
}

void detectEyes(...)
{
  ...
  cv::Rect eyeRect = getLeftmostEye(eyes);
}
```

The `getLeftmostEye` only returns the rect from which the top-left position is leftmost. Nothing serious.

After I got the leftmost eye, I'm going to crop it, apply a histogram equalization to enhance constrat and then the `HoughCircles` function to find the circles in my image.

`eye_detector.cpp`

```cpp
void detectEyes(...)
{
  ...
  cv::Mat eye = face(eyeRect); // crop the leftmost eye
```

```
    cv::equalizeHist(eye, eye);
    std::vector<cv::Vec3f> circles;
    cv::HoughCircles(eye, circles, CV_HOUGH_GRADIENT, 1, eye.cols / 8, 250, 15, eye.rows / 8,
}
```

Let’s take a deep look in what the `HoughCircles` function expects:

- inputImage: The input image
- circles: The circles that it found
- method: Method to be applied
- dp: Inverse ratio of the accumulator resolution
- minDist: Minimal distance between the center of one circle and another
- threshold: Threshold of the edge detector
- minArea: What’s the min area of a circle in the image?
- minRadius: What’s the min radius of a circle in the image?
- maxRadius: What’s the max radius of a circle in the image?

Well, that’s it... As the function itself says, it can detect many circles, but we just want one. So let’s select the one belonging to the eyeball. For that, I chose a very stupid heuristic: Choose the circle that contains more “black” pixels in it! In another words, the circle from which the sum of pixels within it is minimal.

`eye_detector.cpp`

```
cv::Vec3f getEyeball(cv::Mat &eye, std::vector<cv::Vec3f> &circles)
{
  std::vector<int> sums(circles.size(), 0);
  for (int y = 0; y < eye.rows; y++)
  {
      uchar *ptr = eye.ptr<uchar>(y);
      for (int x = 0; x < eye.cols; x++)
      {
          int value = static_cast<int>(*ptr);
          for (int i = 0; i < circles.size(); i++)
          {
              cv::Point center((int)std::round(circles[i][0]), (int)std::round(circles[i][
              int radius = (int)std::round(circles[i][2]);
              if (std::pow(x - center.x, 2) + std::pow(y - center.y, 2) < std::pow(radius,
              {
                  sums[i] += value;
              }
          }
          ++ptr;
      }
  }
  int smallestSum = 9999999;
  int smallestSumIndex = -1;
  for (int i = 0; i < circles.size(); i++)
  {
      if (sums[i] < smallestSum)
      {
          smallestSum = sums[i];
          smallestSumIndex = i;
      }
  }
  return circles[smallestSumIndex];
}

void detectEyes(...)
{
  ...
  if (circles.size() > 0)
  {
      cv::Vec3f eyeball = getEyeball(eye, circles);
  }
}
```

In order to know if a pixel is inside a pixel or not, we just test if the euclidean distance between the pixel location and the circle center is not higher than the circle radius. Piece of cake.

That’s good, now we supposely have the iris. However, the `HoughCircles` algorithms is very unstable, and therefore the iris location can vary a lot! We need to stabilize it to get better results. To do that, we simply calculate the mean of the last five detected iris locations.

`eye_detector.cpp`

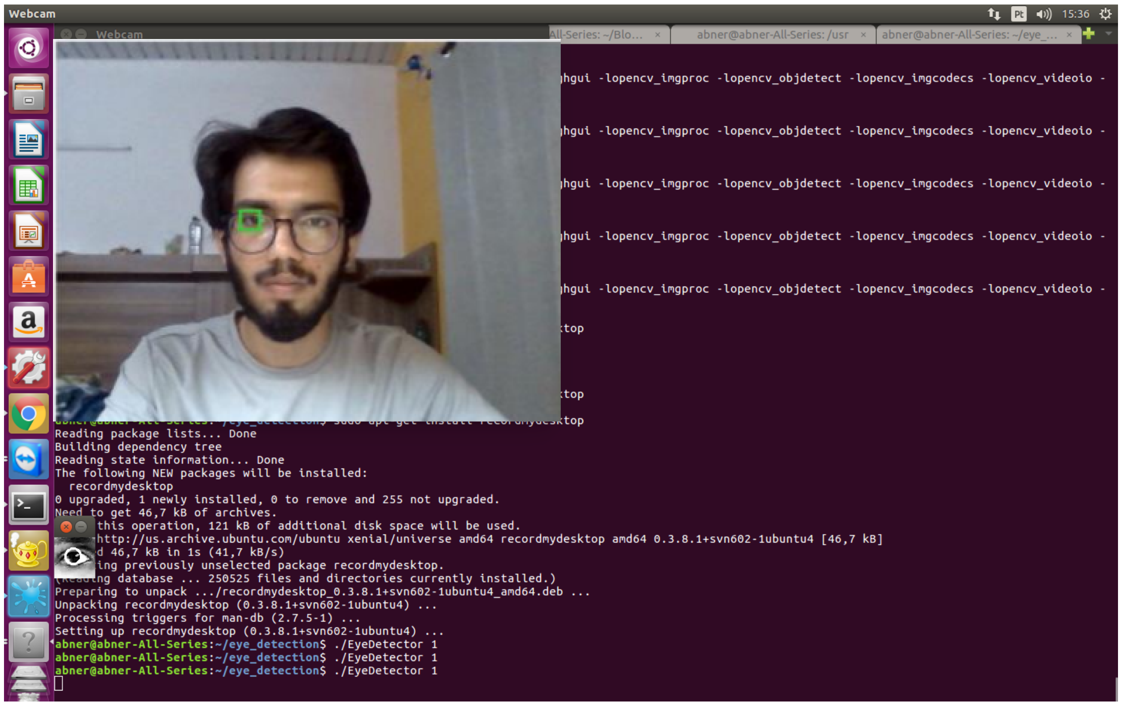
```
std::vector<cv::Point> centers;

cv::Point stabilize(std::vector<cv::Point> &points, int windowSize)
{
  float sumX = 0;
```

```cpp
    float sumY = 0;
    int count = 0;
    for (int i = std::max(0, (int)(points.size() - windowSize)); i < points.size(); i++)
    {
        sumX += points[i].x;
        sumY += points[i].y;
        ++count;
    }
    if (count > 0)
    {
        sumX /= count;
        sumY /= count;
    }
    return cv::Point(sumX, sumY);
}

void detectEyes(...)
{
    ...
    if (circles.size() > 0)
    {
        cv::Vec3f eyeball = getEyeball(eye, circles);
        cv::Point center(eyeball[0], eyeball[1]);
        centers.push_back(center);
        center = stabilize(centers, 5); // we are using the last 5
    }
}
```

Finally, let's draw the iris location and test it!

`eye_detector.cpp`

```cpp
void detectEyes(...)
{
    if (circles.size() > 0)
    {
        ...
        cv::circle(frame, faces[0].tl() + eyeRect.tl() + center, radius, cv::Scalar(0, 0, 25
        cv::circle(eye, center, radius, cv::Scalar(255, 255, 255), 2);
    }
    cv::imshow("Eye", eye);
}
```


Webcam
15:36
Webcam
All-Series: ~/Blo...
abner@abner-All-Series: /usr
abner@abner-All-Series: ~/eye_...

```
hgui -lopencv_imgproc -lopencv_objdetect -lopencv_imgcodecs -lopencv_videoio -

hgui -lopencv_imgproc -lopencv_objdetect -lopencv_imgcodecs -lopencv_videoio -

hgui -lopencv_imgproc -lopencv_objdetect -lopencv_imgcodecs -lopencv_videoio -

hgui -lopencv_imgproc -lopencv_objdetect -lopencv_imgcodecs -lopencv_videoio -

hgui -lopencv_imgproc -lopencv_objdetect -lopencv_imgcodecs -lopencv_videoio -

top

top
top
Reading package lists... Done
Building dependency tree
Reading state information... Done
The following NEW packages will be installed:
  recordmydesktop
0 upgraded, 1 newly installed, 0 to remove and 255 not upgraded.
Need to get 46,7 kB of archives.
   this operation, 121 kB of additional disk space will be used.
   http://us.archive.ubuntu.com/ubuntu xenial/universe amd64 recordmydesktop amd64 0.3.8.1+svn602-1ubuntu4 [46,7 kB]
   d 46,7 kB in 1s (41,7 kB/s)
   ing previously unselected package recordmydesktop.
   ng database ... 250525 files and directories currently installed.)
Preparing to unpack .../recordmydesktop_0.3.8.1+svn602-1ubuntu4_amd64.deb ...
Unpacking recordmydesktop (0.3.8.1+svn602-1ubuntu4) ...
Processing triggers for man-db (2.7.5-1) ...
Setting up recordmydesktop (0.3.8.1+svn602-1ubuntu4) ...
abner@abner-All-Series:~/eye_detection$ ./EyeDetector 1
abner@abner-All-Series:~/eye_detection$ ./EyeDetector 1
abner@abner-All-Series:~/eye_detection$ ./EyeDetector 1
```

Excellent!

## CONTROLLING THE MOUSE

Well, that's something very specific of the operating system that you're using. I'm using Ubuntu, thus I'm going to use `xdotool`. Install xtodo:

```
sudo apt-get install xdotool
```

In xdotool, the command to move the mouse is:

```
xdotool mousemove x y
```

Alright. Let’s just create a variable that defines the mouse position and then set it each time the iris position changes:

eye_detector.cpp

```
cv::Point lastPoint;
cv::Point mousePoint;

void detectEyes(...)
{
  if (circles.size() > 0)
  {
      ...
      if (centers.size() > 1)
      {
          cv::Point diff;
          diff.x = (center.x - lastPoint.x) * 20;
          diff.y = (center.x - lastPoint.y) * -30; // diff in y is higher because it's "ha
      }
      lastPoint = center;
  }
}

void changeMouse(cv::Mat &frame, cv::Point &location)
{
  if (location.x > frame.cols) location.x = frame.cols;
  if (location.x < 0) location.x = 0;
  if (location.y > frame.rows) location.y = frame.rows;
  if (location.y < 0) location.y = 0;
  system(("xdotool mousemove " + std::to_string(location.x) + " " + std::to_string(locatio
}

int main(...)
{
  ...
  while (1)
  {
      ...
      detectEyes(...);
      changeMouse(frame, mousePoint);
      ...
  }
  return 0;
}
```

As you can see, I’m taking the difference of position between the current iris position and the previous iris position. Of course, this is not the best option. Ideally, we would detect the “gaze direction” in relation to difference between the iris position and the “rested” iris position. I let it for you to implement! Not that hard.